Lab 12: Map-Reduce

Instructor: Vivek Sarkar

1 Turning in your lab assignments

We're asking all COMP 322 students to turn in their lab assignments before leaving, as follows:

1. Create a directory called `lab_12/` in your SUGAR account.
2. Do all your work for today's lab in this directory.
3. Before you leave, create a zip file of your work by changing to the parent directory for `lab_12/` and issuing the following command, " `zip -r lab_12.zip lab_12`".
4. Use the turn-in script to submit the contents of the `lab_12.zip` file as a new `lab_12` directory in your turnin directory. (Transfer the file to your CLEAR account if needed.)

2 Update your HJ/DrHJ Installation

NOTE: for nostalgic (and other) reasons, this last COMP 322 lab will be done using HJ.

Please make sure that your HJ installation is current as of March 13, 2012. The performance measurements for today's lab should be done on Sugar, and we've already updated the HJ installation there. If you are also working with a local installation, you can always update it from the HJ download page, `https://wiki.rice.edu/confluence/display/PARPROG/HJDownload`, if needed.

3 Map and Reduce Operations on Sets of Key-Value Pairs

The general idea behind Map-Reduce frameworks was introduced in Lectures 35–36. The *map* function on a collection can be defined as follows, *map* $(f, \{x_1 \ldots x_n\}) = \{f(x_1), \ldots, f(x_n)\}$. Thus, *map* takes two parameters as inputs, a unary function, f, and a collection, L, and returns a new list obtained by applying f to each element in L. It is easy to see that the *map* function is intrinsically parallel since all applications of function f are independent.

The *reduce* operation takes three parameters as inputs in general— a binary function, g, a collection, L, and an *init* value that serves as the identity element. It returns as output a reduced value obtained by applying g on all elements in the collection. For today's lab, we will assume that all functions used for reduce operations are both associative and commutative, hence making them amenable to parallel reductions.

It is convenient to extend the above definitions of *map* and *reduce* operations to *sets* of *key-value* pairs:

- Assume that the input set, L, is of the form $\{(k_1, v_1), \ldots (k_n, v_n)\}$, where each element, (k_i, v_i) consists of a key, k_i, and a value, v_i. Also assume that equality comparison is well defined on all key objects.
- Assume that each application of the map function f generates another set of *intermediate* key-value pairs as follows, $f(k_i, v_i) = \{(k'_1, v'_1), \ldots (k'_m, v'_m)\}$, where each element, (k'_j, v'_j) consists of a key, k'_j, and a value, v'_j. The k'_j keys need not have any direct relationship with the k_i key used in the input of the map function.
- If a function f defined as above is used in a *map* operation, the *map* will generate a set of subsets of key-value pairs. Assume that a *flatten* operation is performed as a post-pass after the *map*.

- Assume that the *reduce* operation takes a set of intermediate key-value pairs, $\{(k'_j, v'_j)\}$ as input, and generates a set of reduced key value sets as output, $\{(k'_j, v''_k)\}$, such that each key, k'_j, appears in at most one pair in the output, and the reduced value v''_k is the result of applying the function g on all values v'_j associated with key k'_j in the input set.

Listing 1 shows how the *WordCount* problem can be solved using map and reduce operations on sets of key-value pairs. All map operations in step a) (line 4) can execute in parallel with only local data accesses, making the map step highly amenable to parallelization. Step b) (line 5) can involve a major reshuffle of data as all key-value pairs with the same key are grouped (gathered) together. Finally, step c) (line 6) performs a standard reduction algorithm for all values with the same key.

```
1 Input: set of words
2 Output: set of (word,count) pairs
3 Algorithm:
4 a) For each input word W, emit (W, 1) as a key-value pair (map step).
5 b) Group together all key-value pairs with the same key (reduce step).
6 c) Perform a sum reduction on all values with the same key (reduce step)↩
     .
```

Listing 1: Computing *Wordcount* using map and reduce operations on sets of key-value pairs

4 HJ's Map-Reduce Framework

As discussed in the lecture, the MapReduce framework has been primarily designed for use in large distributed (warehouse-scale) clusters. However, for simplicity, you will run your Map-Reduce jobs on a single SUGAR node, as in prior lab sessions. Specifically, you will work with a simple MapReduce framework has been developed for HJ clients.

A sample client can be seen in `WordCount.hj`. (The accompanying file, `MapReduce.hj`, will need to be compiled for your lab to work correctly, but won't need to be modified.)

The main features in `WordCount.hj` for using the framework can be found by searching for comments of the form, " MapReduce TODO". As discussed in the lectures, a variety of problems can be solved using the Map-Reduce approach by just changing the body of `map()` and `reduce()` functions.

Your assignment today is twofold:

1. Study the `WordCount.hj` program, and fill in some code in the body of the reduce() function as indicated in the comments labeled 'MapReduce TODO 5".

   Compile the program as follows, `hjc -rt w WordCount.hj`. (Recall that the "`rt -w`" option selects the work-stealing scheduler with a work-first policy.)

   Execute the program on a dedicated SUGAR compute node as follows, "`hj -places 1:8 WordCount words.txt 8 8`". (The last two arguments specify the number of map tasks and reduce tasks respectively.)

   Finally, re-execute the program on one processor as follows, " `hj -places 1:1 WordCount words.txt 1 1`". What difference do you see between the two performance measurements?

2. Copy the `WordCount.hj` program to a new filename, `Index.hj`, and rename the class to `Index`. The goal of this part is to created an inverted index of words as outlined in slide 25 of Lecture 36. Since all words come from one file in this example, the location of a word can just be specified by its start and end positions. Thus, the inverted index should contain one entry per word, and the entry should contain a collection of (start, end) position values to identify multiple locations of the given word.

   Test your solution using the `words.txt` file, as with `WordCount`.